# Using Swift

# with Cocoa and Objective-C

## 苹果官方文档中文翻译版

# Table of Contents

1. 开始
    i. 基本设置
2. 互操作性
    i. 与 Objective-C API 进行交互
    ii. 用 Objective-C 的行为来写 Swift 类
    iii. 使用 Cocoa 数据类型
    iv. 采用 Cocoa 的设计模式
    v. 与 C API 交互
3. 混合与匹配
    i. 在同一个项目各种使用 Swift 和 Objective-C
4. 迁移
    i. 将你的 Objective-C 代码迁移到 Swift 中

# 基本设置

Swift 用于与 Cocoa 和 Objective-C 的无缝兼容。你可以在 Swift 中使用 Ojbective-C 的 API(从系统框架到你自己创建的代码)。这种兼容性让 Swift 成为集成到你 Cocoa 应用开发工作流中一个简单，方便并且强大的工具。

这个手册涵盖了你在开发 Cocoa 应用时的三个重要的兼容性：

- 互操作性 让你可以在 Swift 和 Objective-C 中互相提供接口，允许你在 Objective-C 的代码中使用 Swift 的类并且在 Swift 的代码中使用 Cocoa 中的类，模式和实践。
- 混合和匹配让你能够创建同时包含 Swift 和 Objecitve-C 文件，并且他们之间能够互相调用的项目。
- 迁移可以通过互操作性，混合和匹配很容易的从已有的 Objective-C 代码迁移到 Swift，可以将你用 Objective-C 写的应用的某些部分使用上 Swift 的新特性。

在你开始学习这些特性之前,你需要对如何设置 Swift 环境来使用 Cocoa 系统框架有一个初步了解。

## 设置你的 **Swift** 环境

我们从 Xcode 模版中创建一个基于 Swift 的应用来开始实验在 Swift 中使用 Cocoa 框架。

在 Xcode 中创建 Swift 项目

1. 选择 File > New > Project > (iOS 或 OS X) > Applications > 你选择的模版。
2. 点击 Language 弹出菜单并选择 Swift.

Swift 项目的结构和 Objective-C 项目几乎一样，只有一个重要区别：Swift 没有头文件。 在实现和接口中没有明确的界限，所以关于一个特定类的所有信息都放在一个 **.swift** 文件中。

到这里，你可以在你应用的代理类中写 Swift 代码，或者你可以通过 File > New > File > (iOS 或 OS X) > Other > Swift 来创建一个新的 Swift 类。

## 理解 **Swift** 的导入过程

设置好 Xcode 项目后，你可以将任何框架从 Cocoa 平台导入进来，开始在 Swift 中使用 Objective-C。

所有作为模块使用的 Objective-C 框架(或 C库)都可以直接被导入 Swift。这包括所有的 Objective-C 系统框架－比如 Foundation，UIKit和SpriteKit－也包括系统提供的C语言公用库。例如，如果要引入 Foundation 库，可以像这样简单的在 Swift 头文件中添加这个 import 语句。 `import Foundation` 这个导入语句将所有的 Foundation API 直接导入到 Swift 中 - 包括 NSDate，NSURL，NSMutableData，和他们所有的方法，属性和扩展。

这个导入过程很直观。Objective-C 将 API 都放到头文件中。这些头文件在 Swift 中被编译成了 Objective-C 模块。这些在 Swift 被用作 Swift API。这个导入确定了 Objective-C 中定义的函数， 类，方法和类型如何在 Swift 中展现。对于函数和方法，这个过程影响了它们的参数类型和返回值类型。对于类型，这个导入过程可以做下面这些事：

- 将 Objective-C 中的类型重新映射到 Swift 中，比如 id 映射成 AnyObject。
- 将 Objective-C 中得一些核心类重新映射为 Swift 中得替代类，比如将 NSString 映射到 String。
- 将 Objective-C 中得概念匹配到 Swift 中，比如指针到optional。 在 互操作性* 中，你将会学到更多关于这个映射的内容还有如何将它们引入你的 Swift 代码中。

将 Swift 代码引入到 Objective-C 的方式和把 Objective-C 引入到 Swift 的方式相同。 Swift 提供它的 API—比如通过一个框架—作为 Swift 模块。通过这些 Swift 模块生成 Objective-C 头文件。这些头文件提供了可以映射回 Objective-C 的 API。还有一些 Swift API 是不能够映射到 Objective-C 中的，因为它们使用了 Objective-C 中没有的语言特性。关于在 Objective-C 中使用 Swift 的更多信息，可以查看在同一个项目中使用Swift和Objective-C这章。

> 注意： 你不能将C++代码直接导入到 Swift。可以为 C++ 代码创建 Objective-C 或者 C 的封装。

更多内容请参看 www.theswiftworld.com 网站

# 与 Objective-C API 进行交互

互操作性 是在 Swift 与 Objective-C 中进行交互的一种能力，让你能够访问并且使用另一种语言编写的代码段。在你开始将 Swift 集成到你的应用开发工作流程的时候，你最好能够理解如何能够利用互操作性来重定义，改进，并且加强你写 Cocoa 应用的能力。

互操作性一个重要的方面就是它能让你在写 Swift 代码的时候使用 Objective-C API。到你导入 Objective-C API 后，你可以使用原生的 Swift 语法来初始化这些类，并和他们交互。

## 初始化

要在 Swift 中实例化一个 Objective-C 类，你要通过 Swift 语法来调用它的一个初始化方法。当在 Swift 中调用 Objective-C 的 `init` 方法时,它们使用原生的 Swift 初始化语法。"init" 方法前缀会被去掉并且成为表示这个方法是初始化方法的一个关键字。对于以"initWith" 开头的 `init` 方法，后面的 "With" 也会被去掉。Selector 片段中被去掉的 “init” 或 “initWith” 后面的第一个字母也是小写，并且这个 Selector 片段会被当成第一个参数的名称。 Selector 片段中的其他部分也会作为相应的参数名。每一个 Selector 片段都被包含在括号中并且在调用方法的时候都是必须的。

例如，在 Objective-C 中你可以这样：

```
UITableView _myTableView = [[UITableView alloc] initWithFrame:CGRectZero style:UITableViewStyleGrouped];
```

在 Swift 中你可以这样：

```
let myTableView: UITableView = UITableView(frame: CGRectZero, style: .Grouped)
```

你不需要调用 `alloc` 方法；Swift 能够正确的帮你处理它。注意到当调用 Swift 风格的初始化方法时，“init” 没有出现在任何地方。

你可以在对象初始化的时候明确的制定类型，或者也可以忽略对象类型。 Swift 的类型推断能够正确的决定对象的类型。

```
let myTextField = UITextField(frame: CGRect(x: 0.0, y: 0.0, width: 200.0, height: 40.0))
```

这里面的`UITabeView`和`UITextField` 对象和他们在 Objective-C 中拥有相似的功能，你可以像使用 Objective-C 同样的方式来访问定义在这些类中的属性和方法。

为了一致性和简便性，Objective-C 的工厂方法都映射到了 Swift 中方便的初始化方法。这个映射允许他们被用作想初始化方法一样的简介清晰的语法。例如，在 Objective-C 中，你要像这样调用工厂方法：

```
UIColor *color = [UIColor colorWithRed:0.5 green:0.0 blue:0.5 alpha:1.0];
```

在 Swift 中，你像这样调用：

```
let color = UIColor(red: 0.5, green: 0.0, blue: 0.5, alpha: 1.0)
```

## 访问属性

在 Swift 中使用点号语法来设置 Objective-C 对象的属性。

```
myTextField.textColor = UIColor.darkGrayColor()
```

```
myTextField.text = "Hello world
```

当获取或设置属性的时候，只需要写属性的名字，不需要写括号。但要注意`darkGrayColor`包含了一个括号。这是因为`darkGrayColor`是`UIColor`的一个类方法，不是一个属性。

在 Objective-C 中，返回一个结果并且不带任何参数的方法，可以被当做一个隐式的getter方法，并且可以与访问属性的 getter 方法用相同的语法调用。但这不适用于 Swift。在 Swift 中，只有在 Objective-C 中用 `@property` 语法定义的属性才被当做属性来导入。方法导入和调用的内容在 "使用方法" 这节中详细介绍。

## 使用方法

当在 Swift 中调用 Objective-C 方法的时候，使用点号语法。

当在 Swift 中使用 Objective-C 方法的时候，Objective-C Selector 的第一部分用作方法名并且放在括号外面。第一个参数直接放到括号里面，不需要名称。其他参数部分依照相应的参数名称直接放到括号里面。Selector 中的所有部分在调用的时候都需要。

例如，在 Objective-C 中你这样写：

```
[myTableView insertSubview:mySubview atIndex:2];
```

在 Swift 中，你这样写：

```
myTableView.insertSubview(mySubview, atIndex: 2)
```

如果你调用不带参数的方法，你仍然必须把括号加上。

```
myTableView.layoutIfNeeded()
```

## id 兼容性

Swift 包含一个叫做 `AnyObject` 的协议类型，它代表任何类型的对象，就像 Objective-C 中的 id 一样。 AnyObject 协议能让你在维护灵活的无类型对象的同时，写出类型安全的 Swift 代码。

例如，对于 `id` 类型，你可以对 AnyObject 类型的常量或变量指定任何类型的对象。你也可以对这个变量重新指定其他类型的对象。

```
var myObject: AnyObject = UITableViewCell()
myObject = NSDate()
```

你还能在不进行强制类型转换的情况下调用任何 Objective-C 的方法和访问任何的属性。这包含用 `@objc` 属性标记的 Objective-C 兼容方法。

```
let futureDate = myObject.dateByAddingTimeInterval(10)
let timeSinceNow = myObject.timeIntervalSinceNow
```

然而，因为只有在运行时 AnyObject 类型的对象里面的具体类型才会明确，这样就很容以写出不安全的代码。另外，和 Objective-C 比起来，如果你调用了 Anyobject 中不存在的方法或属性，就会产生运行时错误。例如，如下代码不会产生编译错误，但会在运行时出现未定义的方法错误。

```
myObject.characterAtIndex(5)
```

```
// crash, myObject doesn't respond to that method
```

然而，你可以利用 Swift 中的 Optional 特性来消除你代码中得这些错误。当你在 AnyObject 的对象中调用 Objective-C 方法时，这个方法实际上被当做未封装的 Optional 来调用。你可以像在 AnyObject 中选择性地调用 Optional 方法一样，来使用相同的 optional 链式语法。这个过程同样适用于属性。

例如，在下面列出的代码中，第一行和第二行代码没有被执行，因为 `NSDate` 对象中不存在 `count` 属性和 `characterAtIndex:` 方法。`myLength` 常量被推断为 optional Int 类型，并且设置为 nil。你还可以用 if-let 语句来条件性的解包对象没有响应的方法结果，就像是第三行中得代码那样。

```
let myCount = myObject.count?
let myChar = myObject.characterAtIndex?(5)
if let fifthCharacter = myObject.characterAtIndex(5) {
    println("Found \(fifthCharacter) at index 5")
}
```

在 Swift 中进行向下转型的过程当中，从 AnyObject 到一个更加具体类型的转换，不一定成功并且这个转换会返回一个 optional 类型的值。你可以检测这个 optional 值来决定类型转换是否成功。

```
let userDefaults = NSUserDefaults.standardUserDefaults()
let lastRefreshDate: AnyObject? = userDefaults.objectForKey("LastRefreshDate")
if let date = lastRefreshDate as? NSDate {
    println("\(date.timeIntervalSinceReferenceDate)")
}
```

当然，如果你能确保这个对象所属的类型(并且知道它不是 nil)，你可以强制的调用 `as` 操作符进行转换。

```
let myDate = lastRefreshDate as NSDate
let timeInterval = myDate.timeIntervalSinceReferenceDate
```

## 关于nil

在 Objective-C 中，你使用原始的指针来处理对象的引用，它可以是 `NULL` 值(在 Objective-C 中也被称为 nil)。 在 Swift 中，所有的值，包括对结构体和对象的引用-都确保是非 nil 值。 另外，你可以通过将值得类型包装成 optional 类型来表示那些可以为空得值。当你需要表示一个值缺失，你使用 `nil` 值。关于 optional 的更多信息，可以查看 《Swift编程语言》中 Optionals 这章。

因为 Objective-C 不保证对象一定是非空的， Swift 对引入的 Objective-C API 中所有参数类型中的类，还有返回类型都标记为 Optional。在你使用 Objective-C 对象之前，你要保证它里面不是空的。

在一些情况下，你可能会 绝对 的确定 Objective-C 的方法或属性不会返回一个 `nil` 的对象类型。为了让这种特殊情况下的对象能够更方便的被处理，Swift 引入了 `隐式拆箱的 Optional` 类型。隐式拆箱的 Optional 包含了 Optional 类型的所有安全特性。另外，你在不进行 `nil` 检测和拆箱的情况下，直接的访问这个值。当你访问这种类型的 Optional 的值得时候不进行安全的拆箱操作的话，隐式拆箱的 Optional 会检测这个值是否为空。如果这个值为空，就会产生一个运行时错误。所以，你需要自己对隐式拆箱的 Optional 值进行检测和拆箱，除非你能确定这个值不为空。

## Extension扩展

Swift 中的 Extension Objective-C 的 Category 类似。*Extension* 扩展了现存的类，结构体，枚举的行为，包括那些在 Objective-C 中定义的。你可以在系统框架或是你自定义的类型上定义 Extension。只需要简单的导入相应的模块，并且引用你在 Objective-C 中同样的类，结构体和枚举名称。

例如，你可以通过扩展 `UIBezierPath` 类来基于一个变成和起始点来创建一个等边三角形为形状的简单的贝塞尔曲线。

```
extension UIBezierPath {
    convenience init(triangleSideLength: Float, origin: CGPoint) {
```

```
        self.init()
        let squareRoot = Float(sqrt(3.0))
        let altitude = (squareRoot * triangleSideLength) / 2
        moveToPoint(origin)
        addLineToPoint(CGPoint(x: triangleSideLength, y: origin.x))
        addLineToPoint(CGPoint(x: triangleSideLength / 2, y: altitude))
        closePath()
    }
}
```

你可以使用 Extension 来添加属性(包括类属性和静态属性)。然而，这个数据必须是被计算出来的；Extension 不能为类，结构体和枚举添加存储属性。

这个示例扩展了 `CGRect` 结构体，添加了一个计算出来的 `area` 属性：

```
extension CGRect {
    var area: CGFloat {
    return width * height
    }
}
let rect = CGRect(x: 0.0, y: 0.0, width: 10.0, height: 50.0)
let area = rect.area
// area: CGFloat = 500.0
```

你还可以使用 Extension 来让类在不继承的情况下添加一个协议。如果这个协议是在 Swift 中定义的，你还可以将他们添加到结构体或枚举中，无论是定义在 Swift 中的还是 Objective-C 中的。

你不能用 Extension 来重定义 Objective-C 类型中存在的方法。

## Closure

Objective-C 中得 block 在 Swift 中被引入为 Closure。例如，这里是 Objective-C 中的 block 变量。

```
void (^completionBlock)(NSData *, NSError *) = ^(NSData *data, NSError *error) {/* ... */}
```

这是他们在 Swift 中的样子：

```
let completionBlock: (NSData, NSError) -> Void = {data, error in /* ... */}
```

Swift 中的 Closure 和 Objective-C 中的 block 是相互兼容的，所以你可以将 Swift 的 Closure 传递到需要 block 类型的 Objective-C 方法中。Swift 的 Closure 和函数类型相同，所以你甚至可以将 Swift 中的函数名传入。

Closure 在大部分语义上都和 block 相似，但有一点不同：变量是可修改的，而不是拷贝的。换句话说，在 Swift 中的默认行为相当于 Objective-C 中 __block 关键字。

## 对象比较

在 Swift 中，比较对象有两种方式。第一种，相等性(==)，比较对象直接的内容。 第二种，相同性(===)，确定常量或变量是否引用的相同对象的实例。

在 Swift 中通常用 == 和 === 操作符来比较 Swift 和 Objective-C 对象的相等性。 Swift 对继承自 `NSObject` 的对象提供了一个默认的实现。在这个操作符的实现中， Swift 调用了定义在 `NSObject` 中的 `isEqual:` 方法。 `NSObject` 类仅进行相同性比较，所以你应该对继承自 `NSObject` 的类实现你自己的 `isEqual:` 方法。因为你可以将 Swift 对象(包括哪些不继承自 `NSObject`)传递进 Objective-C API，如果你需要 Objective-C API 比较对象的内容而不是引用，那么你就需要为这些类实现 `isEqual:` 方法。

作为实现相等性的一部分，确保依照对象比较的规则实现 `hash` 属性。另外如果想要将你的类用作字典的键，那么还要使用 `Hashable` 协议并实现 `hashValue` 属性。

## Swift 类型兼容性

当你定义的 Swift 类继承自 `NSObject` 或者其他 Objective-C 类的时候，这个类就自动与 Objective-C 兼容。所有相关这一切的步骤都已经由 Swift 编译器帮你完成。如果你从未在 Objective-C 代码中导入 Swift 类，你不需要对关于类型兼容性的问题着急。另外，如果你的 Swift 类没有继承自 Objective-C 类型并且你还希望在 Objective-C 代码中使用它，那么你可以用下面提到的 `@objc` 属性。

`@objc` 属性能让你的 Swift API 在 Objective-C 和 Objective-C 运行时中可用。换句话说，你可以在任何你想在 Objective-C 代码中用到的 Swift 方法，属性或者类前面加上 `@objc` 属性。如果你的类继承自 Objective-C 类，编译器会帮你加入这个属性。编译器还会将用`@objc` 属性标记的类中的所有属性和方法加上这个`@objc`属性。当你使用 @IBOuntlet,@IBAction,@NSManaged属性的时候，@objc 也会被增加上。当你 Objective-C 的类中使用 selelctor 来实现 target-action 设计模式的时，这个 @objc 属性也很有用- 例如， `NSTimer` 或 `UIButton`。

当你在 Objective-C 中使用 Swift API 的时候，编译器会进行直接的转换。例如，Swift API `func playSong(name:String)` 会在 Objective-C 中被导入成 `- (void) playSong:(NSString *) name`。当你在 Objective-C 中使用 Swift 的初始化方法的时候，编译器会添加方法前面添加 “initWith”，并对之前的初始化方法进行合适的大写处理。例如，Swift 初始化方法 `init (songName: String, artist: String)` 会在 Objective-C 中被导入成 `- (instancetype) initWithSongName:(NSString *) songName artist:(NSString *) artist`。

Swift 提供了一个 `@objc` 的变体，来让你指定你在 Objective-C 中的符号名。例如，如果你的 Swift 类名中包含 Objective-C 中布支持的字符，你可以提供一个在 Objective-C 中到替代名称。如果你要为 Swift 函数提供 Objective-C 名称，要使用 selector 语法。当 selector 中含有参数的时候，记得添加一个冒号(:)

```
@objc(Squirrel)
class Белка {
    @objc(initWithName:)
    init (имя: String) { /*...*/ }
    @objc(hideNuts:inTree:)
    func прячьОрехи(Int, вДереве: Дерево) { /*...*/ }
}
```

当你在 Swift 中使用 `@objc(<#name#>)` 属性的时候，这个类就在 Objective-C 中可用了，不需要任何命名空间。最后，当你将可归档的 Objective-C 类迁移到 Swift 的时候胡这个属性也很有用。因为被归档的类将他们的类名称存储到归档中，你应该使用 `@objc(<#name#>)` 属性来指定和你 Objective-C 类名相同的名称，这样之前的归档就可以在新的 Swift 类中解档。

## Objective-C Selector

Objective-C selector 是一个指向 Objective-C 方法的类型。在 Swift 中，Objective-C Selector 使用 `Selector` 结构来表示。你可以通过字符串值来构造一个 selector，比如 `let mySelector: Selector = “tappedButton:”`。因为字符串值能够自动被转换成 selector，你可以将字符串值传递到任意一个接受 selector 的方法。

```
import UIKit
class MyViewController: UIViewController {
    let myButton = UIButton(frame: CGRect(x: 0, y: 0, width: 100, height: 50))

    init(nibName nibNameOrNil: String?, bundle nibBundleOrNil: NSBundle?) {
        super.init(nibName: nibNameOrNil, bundle: nibBundleOrNil)
        myButton.addTarget(self, action: "tappedButton:", forControlEvents: .TouchUpInside)
    }

    func tappedButton(sender: UIButton!) {
        println("tapped button")
    }
}
```

> 注意 performSelector:方法和它调用的方法不会被导入到 Swift，因为他们是不安全的。

如果你的 Swift 类继承自 Objective-C 类，那么这个类中的所有方法和属性都在 Objective-C 中可用。另外，如果你的 Swift 类没有继承自 Objective-C 类，那么你需要把你想在 Objective-C 中用到的方法用 `@objc` 标记上，就像 《Swift 类型兼容性中提到的》。

更多内容请参看 www.theswiftworld.com 网站

# 用 **Objective-C** 的行为来写 **Swift** 类

互操作性让你能用 Objective-C 的行为定义 Swift 类。你可以在写 Swift 类的时候集成 Objective-C 类，采用 Objective-C 协议，和使用 Objective-C 其他功能的特性。这意味着在 Objective-C 熟悉的基础上用 Swift 现代化并且强大的语言特性来加强它们。

## 继承自 **Objective-C** 类

在 Swift 中，你可以定义 Objective-C 类的子类。要在 Swift 中创建 Objective-C 的子类，在 Swift 类名后面添加一个冒号 (:)，后面紧随着 Objective-C 类的名称。

```
import UIKit

class MySwiftViewController: UIViewController {
    // define the class
}
```

你得到了所有 Objective-C 父类中提供的功能。如果你要提供你对父类方法自己的实现，记得使用 `override` 关键字。

## 采用协议

在 Swift 中，你可以采用定义在 Objective-C 中的协议。就像 Swift 协议，Objective-C 协议也在父类名后面用逗号分隔开。

```
class MySwiftViewController: UIViewController, UITableViewDelegate, UITableViewDataSource {
    // define the class
}
```

如果你要在 Swift 代码中引用 `UITableViewDelegate` 协议，直接用 `UITableViewDelegate` 来引用它(就像在 Objective-C 中用 `id<UITableViewDelegate>` 来引用一样)。

因为 Swift 中没有类的命名空间和协议的定义， Objective-C 中的 `NSObject` 协议在 Swift 中被重新映射为 `NSObjectProtocol` 。

## 创建初始化方法和清理方法

Swift 编译器为了提升你代码的安全性和可预测性，要确保你的初始化方法中对所有的属性都进行了初始化。另外，不像 Objective-C，在 Swift 中没有单独的内存分配供调用方法使用。你要使用原生的 Swift 初始化语法，即便当你使用 Objective-C 类的时候——Swift 将 Objective-C 的初始化方法转化成 Swift 初始化方法。你可以在初始化方法这一章中读到更多实现你自己初始化方法的内容。

当你想在你的类清理的时候执行一些附加的清理工作的时候，你可以实现一个清理方法来代替 `dealloc` 方法。Swift 清理方法会被自动调用，整好在实例销毁之前。 Swift 在调用你子类的清理方法时，也会自动调用父类的清理方法。当你使用 Objective-C 类的时候或者你的 Swift 类继承自 Objective-C 类的时候， Swift 也会为你调用你的父类的 `dealloc` 方法。你可以在清理方法一章中查看更多关于如何自己实现清理方法的内容。

## 与 **Interface Builder** 集成

Swift 编译器包含了为你的 Swift 类打开 interface builder 特性的属性。就像在 Objective-C 中， 你可以在 Swift 中使用 outlet，action， live rendering。

## 使用 **Outlet** 和 **Action**

Outlet 和 action 能让你将你的代码与 Interface Builder 中的用户界面进行关联。要在 Swift 中使用 outlet 和 action，需要在属性和方法定义的前面加上 `IBOutlet` 或 `IBAction` 。同样使用 `IBOutlet` 来定义 outlet 集合——只需要将类型指定为

array。

当你在 Swift 中定义一个 outlet，编译器自动会将这个类型转换成隐式拆包后的 Optional 值，而且是弱引用的，并为它赋初始值为 nil。实际上，编译器将 `IBOutlet var name: Type` 替换为 `IBOutlet weak var name: Type! = nil`。编译器已经将它转换为隐式拆包的 Optional 类型，这样就不必一定要在初始化方法中为它指定初始值。因为在从 storyboard 或 xib 文件初始化后，你可以假设 outlet 已经被连接上。Outlet 默认是 weak 引用，你创建的 outlet 默认都是弱引用关系。

例如，如下的 Swift 代码中的类，定义了一个 outlet，一个 outlet 集合，还有一个 action：

```
class MyViewController: UIViewController {
    @IBOutlet var button: UIButton
    @IBOutlet var textFields: [UITextField]
    @IBAction func buttonTapped(AnyObject) {
        println("button tapped!")
    }
}
```

因为 `buttonTapped:` 方法传入的参数没有被使用，参数名就可以忽略掉。

## Live Rendering

你可以使用两个不同的属性——@IBDesignable 和 @IBInspectable——来打开在 Interface Builder 设计的实时交互性的自定义视图。当你创建一个继承自 `UIView` 和 `NSView` 的自定义视图，你可以在类定义前面加上 `@IBDesignable` 属性。当你将自定义视图添加到 Interface Builder 后(在 inspector 面板中设置视图的自定义类)，Interface Builder 在 canvas 上渲染你的视图。

Live rendering 只能从导入的框架中使用。

你还可以为类的属性添加 `@IBInspectable` 属性。在你将自定义视图添加到 Interface Builder 后，你可以在 inspector 中编辑这些属性。

```
@IBDesignable
class MyCustomView: UIView {
    @IBInspectable var textColor: UIColor
    @IBInspectable var iconHeight: CGFloat
    /* ... */
}
```

## 指定类属性的属性

在 Objective-C 中，类的属性有一系列的特性可以对属性的行为指定一些附加信息。在 Swift 中，你用不同的方式来指定这些特性。

### Strong 和 Weak

Swift 的属性默认情况下是 strong 的。使用 `weak` 关键字来表示属性是一个指向存储为它的值的对象的弱引用。这个关键字只适用于那些 Optional 类型的属性。关于更多信息，可以查看 Attributes 一章。

### 可读可写和只读

在 Swift 中，没有 `readwrite` 和 `readonly` 属性。当定义一个存储的属性时，`let` 表示只读，`var` 表示可读可写。当定义一个计算出来的属性时，只提供 getter 方法，它就是只读的，同时提供 getter 和 setter 方法，它就是可读可写的。更多信息，可以查看 Properties 一章。

### Copy 语法

在 Swift 中， Objective-C 的 `copy` 类型的属性被翻译为 `@NSCopying`。属性的类型必须遵从 `NSCopying` 协议。关于更多的信息，请查看 Attributes 一章。

## 实现 Core Data Managed Object 的子类

Core Data 提供了 `NSManagedObject` 子类属性的底层存储和实现。为你在 Core Data 数据模型中相应的属性或关系的子类的每一个类属性前面加上 `@NSManaged` 标记。就像 Objective-C 中的 `@dynamic` 标记，`@NSManaged` 告诉 Swift 编译器，一个属性的存储和实现将在运行时提供。然而，不像 `@dynamic`，`@NSManaged` 标记，只有在 Core Data 的支持中可用。

Swift 的类是用命名空间管理的——他们用他们编译进的模块来区分(一般情况下是项目)。要使用 Swift 对 `NSManagedObject` 的子类到你自己的 Core Data 模型中，在 Class 文本框中加入你模型的前缀。

更多内容请参看 www.theswiftworld.com 网站

# 使用 Cocoa 数据类型

作为 Objective-C 互操作性的一部分，Swift 提供了方便和高效的方式来处理 Cocoa 数据类型。

Swift 自动将 Objective-C 类型转换成 Swift 类型，并且将一些 Swift 类型转换成 Objective-C 类型。还有很多数据类型在 Swift 和 Objective-C 中是可以通用的。这些可转换或者可通用的数据类型被叫做 *bridged* 数据类型。例如，在 Swift 代码中，你可以将 `Array` 类型的值传递给接受 `NSArray` 类型对象的方法。当你使用 `as` 来转换这些类型 - 或明确的提供常量或变量的类型 - Swift 会桥接数据类型。

Swift 还提供对 Foundation 数据类型方便的接口，让你与其他 Swift 代码语言感觉自然和统一的方式来使用它们。

## Strings

Swift 自动对 `String` 和 `NSString` 类进行桥接。这意味着任何你使用 `NSString` 的地方，你都可以使用 Swift 的 `String` 类型来替代并且能够获得这两种类型的所有特性—— `String` 类型的插值和为 Swift 设计的 API，还有 `NSString` 类广泛的功能。由于这个原因，你应该几乎不需要在你的代码中直接使用 `NSString` 。事实上，当 Swift 导入 Objective-C API 的时候，它将所有的 `NSString` 类型替换成 `String` 类型。当你的 Objective-C 代码使用 Swift 类的时候，导入器将所有的 `String` 类型替换成 `NSString` 类型。

要打开 String 桥接，只需要引入 Foundation 库。例如，你可以在 Swift 字符串中调用 `NSString` 类的 `capitalizedString` 方法，并且 Swift 会自动将 Swift 的 `String` 类桥接到 `NSString` 对象并调用这个方法。这个方法甚至还可以返回 Swift 类型，因为他在导入过程中被转换了。

```
import Foundation
let greeting = "hello, world!"
let capitalizedGreeting = greeting.capitalizedString
// capitalizedGreeting: String = Hello, World!
```

如果你需要使用 `NSString` 对象，你可以通过类型转换将它转换为 Swift 的 `String` 类型。 `String` 类型总能够从 `NSString` 转换到 Swift 的 `String` ，所以你不需要使用 `as?` 的 Optional 版本来进行转化。你还可以通过直接输入常量和变量来从字符串字面值中创建一个 `NSString` 对象。

```
import Foundation
let myString: NSString = "123"
if let integerValue = (myString as String).toInt() {
    println("\(myString) is the integer \(integerValue)")
}
```

## 本地化

在 Objective-C 中，你使用 `NSLocalizedString` 宏系列来对字符串进行本地化，这组宏包括 `NSLocalizedString` ， `NSLocalizedStringFromTable` ， `NSLocalizedStringFromTableInBundle` ，和 `NSLocalizedStringWithDefaultValue` 。在 Swift 中你可以使用单独一个函数来完成和上面那些同样的功能—— `NSLocalizedString(key:tableName:bundle:value:comment:)` 。 `NSLocalizedString` 函数提供了 `tableName` ， `bundle` ， `value` 参数的默认值。

## Numbers

Swift 自动地桥接原生的 number 类型，比如 `Int` ， `Float` ， `NSNumber` 。这样的桥接让你从这些类型中创建 `NSNumber` 。

```
let n = 42
let m: NSNumber = n
```

它还允许你传入 `Int` 值。例如，对于那些 `NSNumber` 类型的参数。然而，因为 `NSNumber` 可以包含一些列不同的类型，你不

能将他们传递给那些需要 `Int` 类型的值。

所有下面这些类型都自动桥接到 `NSNumber` ：

- Int
- UInt
- Float
- Double
- Bool

## 集合类

Swift 自动将 `NSArray` 和 `NSDictionary` 类桥接到他们的原生 Swift 实现中。这意味着你可以使用 Swift 强大的算法和自然的语法来操作集合——并可以互换的使用 Foundation 和 Swift 集合类型。

## 数组

Swift 自动对 `Array` 和 `NSArray` 进行桥接。当你将 `NSArray` 桥接到 Swift 数组中时，最终得到的数组是 `[AnyObject]` 类型。 `Objective-C` 或 `Swift` 类都是和 `AnyObject` 兼容的，或者对象可以桥接到其中一个。因为 Objective-C 对象是和 `AnyObject` 兼容的，所以你可以将任何的 `NSArray` 对象桥接到 Swift 数组中。因为所有的 `NSArray` 对象都可以桥接到 Swift 数组中，Swift 编译器在导入 `Objective-C` API的时候，将所有的 `NSArray` 类替换成 `[AnyObject]`。

在你将 `NSArray` 对象桥接到 Swift 数组中后，你还可以将数组 向下转型 到更具体的类型。与将 `NSArray` 转换成 `[NSArray]` 类型不一样，从 `AnyObject` 转换到具体的类型不保证成功。编译器知道运行时才能确定知道数组中所有的元素是否能够转换为你指定的类型。所以，从 `[AnyObject]` 向下转型到 `[SomeType]` 会返回一个 Optional 值。例如，如果你知道 Swift 数组仅包含 `UIView` 类的实例(或 `UIView` 的子类)，你可以将数组中的 `AnyObject` 类型元素转换成 `UIView` 对象。如果 Swift 数组中的任意元素在运行时不是 `UIView` 类型，这个转换会返回 `nil`。

```
let swiftArray = foundationArray as [AnyObject]
if let downcastedSwiftArray = swiftArray as? [UIView] {
    // downcastedSwiftArray contains only UIView objects
}
```

你可以在一个 for 循环中，直接从 `NSArray` 对象向下转型到包含指定类型的数组：

```
for aView: UIView! in foundationArray {
    // aView is of type UIView
}
```

另外，你可以对你正在遍历的数组进行向下转型——这两种风格的结果都是一样的：

```
for aView in foundationArray as [UIView] {
    // aView is of type UIView
}
```

> 这种转换是强制转换，如果转换不成功，将会产生运行时错误。

当你将 Swift 数组转换成 `NSArary` 对象的时候，Swift 数组中得元素必须是与 `AnyObject` 兼容的。例如，一个包含 `Int` 类型元素的 `[Int]` 类型的数组。 `Int` 类型不是类的一个实例，但因为 `Int` 类型和 `NSNumber` 可以桥接，这个 `Int` 类型是和 `Anyobject` 兼容的。这样你可以将 `[Int]` 类型的 Swift 数组桥接到 `NSArray` 对象上面。如果 Swift 数组中的一个元素不是 `AnyObject` 兼容的，当你桥接到 `NSArray` 对象的时候，就会产生一个运行时错误。

你还可以从 Swift 字面值中直接创建 `NSArray` 对象，就像下面代码中描述的那样。当你明确的将常量或变量指定为 `NSArray` 类型的对象并给他赋一个数组字面值，Swift 会创建一个 `NSArray` 对象而不是 Swift 数组。

```
let schoolSupplies: NSArray = ["Pencil", "Eraser", "Notebook"]
// schoolSupplies is an NSArray object containing NSString objects
```

在上面的例子中，Swift 数组字面值包含了 `String` 字面值。因为 `String` 类型桥接到 `NSString` 类中，数组字面值被桥接到 `NSArray` 对象并赋值到 `schoolSupplies` 对象中。

当你在 Objective-C 代码中使用 Swift 类或协议的时候，导入器将它导入到所有 Swift 数组转换成 NSArray。如果你将 `NSArray` 传递给需要不同类型参数的 Swift 方法，会产生一个运行时错误。如果 Swift API 返回的 Swift 数组不能被桥接到 `NSArray`,会产生一个运行时错误。

## 字典

除了数组，Swift 还自动对 `Dictionary` 和 `NSDictionary` 类型之间进行自动转换。当你将 `NSDictionary` 类桥接到 Swift 字典中，返回的字典类型是 [NSObject: AnyObject]。你可以将任意的 `NSDictionary` 对象桥接到 Swift 字典上，因为所有的 Objective-C 对象都是与 `AnyObject` 兼容的。回忆一下，一个 Objective-C 或 Swift 类的实例是和 `AnyObject` 兼容的，或者它可以桥接到其中一个。所有的 `NSDictionary` 对象都可以桥接到 Swift 字典中，这样 Swift 编译器当在导入 Objective-C API 的时候，会将 `NSDictionary` 类替换成 `[NSObject: AnyObject]`。比如，当你在 Objective-C 代码中使用 Swift 类或协议，导入器将会把与 Objective-C 兼容的 Swift 字典，重新映射成 `NSDictionary` 对象。

在你将 `NSDictionary` 对象桥接到 Swift 字典后，你还可以将这个字典向下转型到更具体的类型。就像向下转型 Swift 数组，向下转型 Swift 字典不保证成功。将 `[NSObject: AnyObject]` 进行向下转到更具体类型的返回结果是一个 Optional 值。

当你以反方向进行转型的时候——从 Swift 字典转换到 `NSDictionary` 对象——键和值必须是类的一个实例，或可桥接到一个类的实例。

你还可以直接从 Swift 字典字面值中创建一个 `NSDictionary` 对象，依照上述的桥接规则。当你明确地创建一个 `NSDictionary` 常量或变量，并把他们赋值给一个字典字面值的时候，Swift 会创建一个 `NSDictionary` 对象，而不是 Swift 字典。

## Foundation 数据类型

Swift 提供一个方便的接口用于覆盖定义在 Foundation 框架中的数据类型。使用这个覆盖用于类似 `NSSize` 和 `NSPoint` 这样的数据类型，通过和 Swift 语言通用的语法。例如，你可以用这样的语法来创建 `NSSize` 结构：

```
let size = NSSize(width: 20, height: 40)
```

这个覆盖还能让你用自然的方式调用在结构体上的 Foundation 函数。

```
let rect = CGRect(x: 50, y: 50, width: 100, height: 100)
let width = rect.width    // equivalent of CGRectGetWidth(rect)
let maxX = rect.maxY      // equivalent of CGRectGetMaxY(rect)
```

Swift 将 `NSUInteger` 和 `NSInteger` 桥接到 `Int`。这两个类型在 Foundation API 中，都以 `Int` 形式存在。 `Int` 在 Swift 用作一致性，但如果你需要无符号整形数据的话，也可以使用 `UInt`。

## Foundation 函数

`NSLog` 可以用于在系统控制台中输出日志。使用和 `Objective-C` 中同样的语法。

```
NSLog("%.7f", pi)         // Logs "3.1415927" to the console
```

然而，Swift 还有像是 `print` 和 `println` 这样的函数。这些函数因为 Swift 的字符串插值变得简单强大。他们不会打印到控制台中，但可以用作通用的打印需要。

`NSAssert` 函数没有转移到 Swift 中，你需要使用 `assert` 函数。

## Core Foundation

Core Foundation 类型会自动导入到 Swift 类中。无论内存管理标记是否已经提供， Swift 自动管理 Core Foundation 对象的内存，包括你自己实例化的 Core Foundation 对象。在 Swift 中，你可以交替使用两个方向的 Foundation 和 Core Foundation 类型。你还可以将一些自动桥接的 Core Foundation 类型桥接到 Swift 标准库类型。

### 重新映射类型

当 Swift 导入 Core Foundation 类型，编译器重新映射这些类型。编译器会删除每个类型名称后面的 *Ref*，因为所有 Swift 类都是引用类型，这样的后缀就多余了。

Core Foundation 的 `CFTypeRef` 完全映射到 `AnyObject` 类型。在你用到 `CFTypeRef` 的任何地方，你应该在你的代码中使用 `AnyObject`。

### 内存管理对象

Core Foundation 返回的被标记的 API 在 Swift 中会进行自动的内存管理 —— 你不需要自己调用 `CFRetain`， `CFRelease`，或 `CFAutorelease`。如果你从自己的 C 函数和 Objective-C 方法中返回 Core Foundation 对象，将他们标记为 `CF_RETURNS_RETAINED` 或 `CF_RETURNS_NOT_RETAINED`。编译器在编译 Swift 代码时会自动为他们加上内存管理相关的调用。如果你只是使用标记 API，没有简介返回 Core Foundation 对象，你可以跳过后面的章节。反之，继续去学习关于操作未被管理的 Core Foundation 对象。

### 未被管理的对象

当 Swift 导入没有被标记的 API 时，编译器就不能对返回的 Core Foundation 对象进行自动内存管理。Swift 将这些返回的 Core Foundation 对象封装进 `UNmanaged<T>` 结构中。虽有间接返回的 Core Foundation 对象也都是未被管理的。例如，这里是未被标记的 C 函数：

```
CFStringRef StringByAddingTwoStrings(CFStringRef string1, CFStringRef string2)
```

这里是如何导入到 Swift 中：

```
func StringByAddingTwoStrings(CFString!, CFString!) -> Unmanaged<CFString>!
```

当你从未标记的 API 中接受到一个未被管理的对象时，你应该在使用它之前把他转换成被内存管理的对象。这样，Swift 就可以帮你处理内存管理的问题。 `Unmanaged<T>` 结构提供了两个方法将未管理的对象转换成内存管理的对象 —— `takeUnretainedValue()` 和 `takeRetainedValue()`。这两个方法都返回原始并且未封装类型的对象。你通过你调用的 API 是否返回 retain 的对象来选择调用其中哪个方法。

例如，假如上面的 C 函数在返回 `CFString` 之前没有进行 retain 操作。要开始使用这个对象，你要使用 `takeUnretainedValue()` 函数。

```
let memoryManagedResult = StringByAddingTwoStrings(str1, str2).takeUnretainedValue()
// memoryManagedResult is a memory managed CFString
```

你还可以调用 `retain()`, `release()` 和 `autorelease()` 方法，但这个方式是不推荐的。

更多内容请参看 www.theswiftworld.com 网站

更多内容请参看 http://www.theswitftworld.com 网站

# 采用 Cocoa 的设计模式

使用 Cocoa 已有的设计模式，可以帮助你建立良好设计，并且有弹性的 App。许多这些模式都依赖于定义在 Objective-C 中的类。因为 Swift 和 Objective-C 之间的互操作性，你可以在 Swift 中利用这些公用的模式。在很多案例中，你可以使用 Swift 的语言特性来扩展或简化现有的 Cocoa 设计模式，让他们更加强大并易于使用。

## 代理

在 Swift 和 Objective-C 中，代理通常都以协议的方式来表示。和 Objective-C 比起来，当你在 Swift 中实现代理时，设计模式还是相同的，但是实现方式改变了。就像 Objective-C，在你发消息给代理之前，你回检测它是否为 `nil` —— 还有这个方法是否为 Optional 的，你检查这个代理是否响应这个 selector。在 Swift 中，可以通过维护类型安全来解决这个问题。下面列出的代码描述了如下的过程：

1. 检测 `myDelegate` 为非 nil。
2. 检测 `myDelegate` 是否实现了 `window:willUseFullScreenContentSize:` 。
3. 如果 1 和 2 的条件满足，那么调用这个方法并将返回值赋给 `fullScreenSize` 变量。
4. 打印方法的返回值。

```
// @interface MyObject : NSObject
// @property (nonatomic, weak) id<NSWindowDelegate> delegate;
// @end
if let fullScreenSize = myDelegate?.window?(myWindow, willUseFullScreenContentSize: mySize) {
    println(NSStringFromSize(fullScreenSize))
}
```

> 注意：在一个纯 Swift 写的 App 中，输入 `delegate` 作为 Optional 对象 > `NSWindowDelegate` 的属性，并且为他赋初值 `nil` 。

## 懒加载

信息即将来临，你可以在 Swift 编程语言一书中，读到更多关于懒存储属性的内容。

## 错误报告

Swift 中的错误报告和 Objective-C 中的一样，并且附加上 Optional 返回值提供的附加特性。在最简单的例子中，你从函数中返回一个 `Bool` 类型的值来表示是否成功。当你需要报告错误的原因，你可以给函数添加一个 `NSErrorPointer` 类型的输出参数。这个类型类型几乎等于 Objective-C 的 `NSError **` 类型，并且更加的内存安全和支持 Optional 类型。你可以通过对 Optional 的 `NSError` 类型添加 `&` 前缀操作符，就像下面的代码列出的那样：

```
var writeError : NSError?
let written = myString.writeToFile(path, atomically: false,
    encoding: NSUTF8StringEncoding,
    error: &writeError)
if !written {
    if let error = writeError {
        println("write failure: \(error.localizedDescription)")
    }
}
```

当你自己实现需要配置 `NSErrorPointer` 对象的函数，你将 `NSErrorPointer` 对象的 `memory` 属性设置为你创建的 `NSError` 对象。确保调用者首先传递了非 `nil` 的 `NSErrorPointer` 对象。

```
func contentsForType(typeName: String!, error: NSErrorPointer) -> AnyObject! {
    if cannotProduceContentsForType(typeName) {
        if error {
            error.memory = NSError(domain: domain, code: code, userInfo: [:])
        }
```

```
            return nil
        }
        // ...
    }
```

## Key-Value 检测

信息即将到来。

## Target-Action

Target-action 是 Cocoa 一个公用的设计模式，用于当一个指定事件发生时，一个对象向另一个对象发送消息。这个 target-action 模型在 Swift 和 Objective-C 中基本相似。在 Swift 中，你可以使用 `Selector` 类型来引用 Objective-C 的 selector。关于在 Swift 中使用 target-action 的例子，可以查看 Objective-C Selector 这章。

## 内省

在 Objective-C 中，你使用 `isKindOfClass:` 来检测一个对象是否为某些类型，使用 `conformsToProtocol:` 方法来检测一个对象是否遵守一个特定协议。在 Swift 中，你使用 `is` 操作符来检测类型，后者使用 `as?` 操作符来进行向下转型。

你可以通过 `is` 检测一个实例是否为某些类的子类。如果这个实例是子类的类型， `is` 操作符会返回 `true` , 如果不是，会返回 `false` 。

```
if object is UIButton {
    // object is of type UIButton
} else {
    // object is not of type UIButton
}
```

你还可以试着使用 `as?` 操作符来向下转型到子类。 `as?` 操作符返回一个 Optional 值，这个值可以使用 `if-let` 语句绑定到一个常量上面。

```
if let button = object as? UIButton {
    // object is successfully cast to type UIButton and bound to button
} else {
    // object could not be cast to type UIButton
}
```

关于更多信息，可以查看 Swift 变成语言中类型转换这章。

检测并转换到某个协议上与检测和转换到某个类上的语法相同。这里是使用 `as?` 来检测协议兼容性的一个例子：

```
if let dataSource = object as? UITableViewDataSource {
    // object conforms to UITableViewDataSource and is bound to dataSource
} else {
    // object not conform to UITableViewDataSource
}
```

注意在这个转换后， `dataSource` 常量为 `UITableViewDataSource` 类型，这样你只能调用 `UITableViewDataSource` 中得方法和属性。如果你要进行其他操作，你必须把他转换成另外的类型。 关于更多的信息，可以查看 Swift 编程语言中协议一章 ` 更多内容请参看 www.theswiftworld.com 网站

# 与 C API 交互

作为与 Objective-C 互操作性的一部分，Swift 维护了与一定数量的 C 语言类型和特性的兼容性。Swift 还提供了与 C 构造器和设计模式交互的方式。

## 原始类型

Swift 提供了和 C 原生整型的等值-例如， `char` ， `int` ， `float` 和 `double` 。然而，没有在这些类型和 Swift 整型类型之间的隐式转换，比如 `Int` 。因此，如果你的代码实在需要它们，才使用这些类型，否则在任何可能的情况下，都使用 `Int` 。

| 类型 | **Swift 类型** |
| --- | --- |
| bool | CBool |
| char, signed char | CChar |
| unsigned char | CUnsignedChar |
| short | CShort |
| unsigned short | CUnsignedShort |
| int | CInt |
| unsigned int | CUnsignedInt |
| long | CLong |
| unsigned long | CUnsignedLong |
| long long | CLongLong |
| unsigned long long | CUnsignedLongLong |
| wchart | CWideChar |
| char16t | CChar16 |
| char32t | CChar32 |
| float | CFloat |
| double | CDouble |

## 枚举

Swift 会导入所有用 `NS_ENUM` 宏来标记的 C 风格的枚举作为 Swift 枚举。这意味着，枚举值名称的前缀，在导入到 Swift 中得时候，会被截断，无论他们是定义在系统框架中，或是自定义代码中。例如，看一下这个 Objective-C 枚举：

```
typedef NS_ENUM(NSInteger, UITableViewCellStyle) {
   UITableViewCellStyleDefault,
   UITableViewCellStyleValue1,
   UITableViewCellStyleValue2,
   UITableViewCellStyleSubtitle
};
```

在 Swift 中，它被导入为这样的值：

```
enum UITableViewCellStyle: Int {
    case Default
    case Value1
    case Value2
    case Subtitle
}
```

当你引用枚举值，使用点号(.)后面加上枚举值名称的形式。

```
let cellStyle: UITableViewCellStyle = .Default
```

Swift 还会导入用 `NS_OPTIONS` 宏来标记的选项。选项的行为类似于导入的枚举，选项还可以支持一些位运算，比如 `&`,`|`,和 `~`。在 Objective-C 中，你可以用 0 来代表空的选项集。在 Swift 中使用 `nil` 来表示缺少的选项。

## 指针

Swift 尽量避免让你直接访问到指针。然而，当你需要直接访问内存的时候，也有几个不同种类的指针供你使用。下面的表格，使用 `Type` 来作为类型名称的占位符，用于指示语法的映射。

对于参数，下面的映射适用：

| C 语法 | Swift 语法 |
| --- | --- |
| const void * | CConstVoidPointer |
| void * | CMutableVoidPointer |
| const Type * | CConstPointer<Type> |
| Type * | CMutablePointer<Type> |

对于多于一级指针深度的返回类型，变量和参数类型，适用于如下的映射：

| C 语法 | Swift 语法 |
| --- | --- |
| void * | COpaquePointer |
| Type * | UnsafePointer<Type> |

对于 class 类型，适用下面的映射：

| C 语法 | Swift 语法 |
| --- | --- |
| Type * const * | UnsafePointer<Type> |
| Type * __strong * | UnsafeMutablePointer <Type> |
| Type ** | AutoreleasingUnsafePointer <Type> |

## C Mutable Pointers

当函数定义为接受 `CMutablePointer<Type>` 类型的参数时，他可以接受一下参数的任意一种：

- `nil`，代表空指针。
- CMutablePointer 值
- 输入-输出表达式(可被保存修改的参数),操作数被存储在 `Type` 类型的左值中，并使用左值的地址传入。
- 输入-输出 `Type` 类型的值，作为指向数组首元素的指针传入，并在调用的时间中是延长生存周期的。

如果你将函数定义为下面这样：

```
func takesAMutablePointer(x: CMutablePointer<Float>) { /*...*/ }
```

你可以以下面任意的方式调用它：

```
var x: Float = 0.0
var p: CMutablePointer<Float> = nil
```

```
var a: [Float] = [1.0, 2.0, 3.0]
takesAMutablePointer(nil)
takesAMutablePointer(p)
takesAMutablePointer(&x)
takesAMutablePointer(&a)
```

当一个函数被定义为接受 `CMutableVoidPointer` 类型的参数，他也可以接受 `CMutablePointer%lt;Type%gt;` 形式定义的任意 `Type` 类型的参数。

如果你这样定义一个函数：

```
func takesAMutableVoidPointer(x: CMutableVoidPointer)  { /* ... */ }
```

你可以用以下任意方式调用它：

```
var x: Float = 0.0, y: Int = 0
var p: CMutablePointer<Float> = nil, q: CMutablePointer<Int> = nil
var a: [Float] = [1.0, 2.0, 3.0], b: Int = [1, 2, 3]
takesAMutableVoidPointer(nil)
takesAMutableVoidPointer(p)
takesAMutableVoidPointer(q)
takesAMutableVoidPointer(&x)
takesAMutableVoidPointer(&y)
takesAMutableVoidPointer(&a)
takesAMutableVoidPointer(&b)
```

## C 常量指针

当一个函数定义为使用 `CConstPointer%lt;Type%gt;` 类型的参数，他可以接受如下形式的传入：

- `nil` ,作为空指针传入 * `CMutablePointer<Type>` , `CMutablePointer<Type>` , `CMutablePointer<Type>` , `CMutablePointer<Type>` , 或 `CMutablePointer<Type>` 的值，在需要的时候会被转换成 `CConstPointer<Type>`
- 左值为 `Type` 类型的输入-输出表达式，用左值的地址作为参数传递。
- 一个 `[Type]` 值，指向数组首元素的指针传递进来，并在调用过程中声明周期进行扩展。

如果你像这样定义了函数：

```
func takesAConstPointer(x: CConstPointer%lt;Float%gt;) { /*...*/ }
```

你可以用以下任意方式调用它：

```
var x: Float = 0.0
var p: CConstPointer%lt;Float%gt; = nil
takesAConstPointer(nil)
takesAConstPointer(p)
takesAConstPointer(&x)
takesAConstPointer([1.0, 2.0, 3.0])
```

当函数定义为接受 `CConstVoidPointer` 类型的参数时，它可以接受用 `CConstPointer%lt;Type%gt;` 形式定义的任意 `Type` 类型。

如果你像这样定义了函数：

```
func takesAConstVoidPointer(x: CConstVoidPointer)  { /* ... */ }
```

你可以以下面任意形式调用它：

```
var x: Float = 0.0, y: Int = 0
var p: CConstPointer<Float> = nil, q: CConstPointer<Int> = nil
takesAConstVoidPointer(nil)
takesAConstVoidPointer(p)
takesAConstVoidPointer(q)
takesAConstVoidPointer(&x)
takesAConstVoidPointer(&y)
takesAConstVoidPointer([1.0, 2.0, 3.0])
takesAConstVoidPointer([1, 2, 3])
```

## AutoreleasingUnsafePointer

当一个函数定义为接受 `AutoreleasingUnsafePointer<Type>` 类型的参数，他接受下面任意一种：

- `nil` 作为空指针传入。
- AutoreleasingUnsafePointer%lt;Type%gt; 类型的值。
- 输入-输出表达式，操作符是原始类型的拷贝。这个缓冲区的地址被传入被调者，并且在返回的时候，这个缓冲区中得值会被调用，retain，并且重新赋值到操作值中。

注意这个列表中不包含数组。

如果你像这样定义函数：

```
func takesAnAutoreleasingPointer(x: AutoreleasingUnsafePointer<NSDate?>) { /* ... */ }
```

你可以用如下的方式调用它:

```
var x: NSDate? = nil
var p: AutoreleasingUnsafePointer<NSDate?> = nil
takesAnAutoreleasingPointer(nil)
takesAnAutoreleasingPointer(p)
takesAnAutoreleasingPointer(&x)
```

注意 C 函数指针没有引入到 Swift 中。

## 全局常量

在 C 和 Objective-C 源文件中定义的全局常量会被 Swift 编译器自动导入为 Swift 全局常量。

## 预处理指令

Swift 编译器中不包含预处理器，而是利用编译时属性，构建参数和语言特性来完成同样的功能。因此，预处理指令没有被导入到 Swift 中。

## 简单宏

在 C 和 Objective-C 中，你使用 `#define` 指令来定义原始常量，而在 Swift 中，你使用全局常量来代替。例如，常量定义 `#define FADE_ANIMATION_DURATION 0.35` 可以在 Swift 中用 `let FADE_ANIMATION_DURATION = 0.35` 来更好的表示。因为这样类似常量的简单宏定义会直接映射为 Swift 全局常量，编译器会自动导入定义在 C 或 Objective-C 源文件中的简单宏。

## 复合宏

在C 和 Objective-C 中使用的复合宏没有 Swift 中与之对应的东西。复合宏是那些不定义常量，而是用括号进行类似函数功能的宏。你在 C 和 Objective-C 中使用复合宏定义来避免类型检测的限制，或者避免重复输入大量的模板代码。然而，宏让调试和重构变得困难。在 Swift 中，你可以使用函数和泛型来取得同样的结果，而不需要进行任何的折中操作。这样，在 C 和 Objective-C 源文件中的复合宏，在你的 Swift 代码中是不可用的。

## 构建配置

Swift 和 Objective-C 代码会条件性的用不同的方式编译。Swift 代码会条件性地根据 构建配置 的结果值进行配置。构建配置包括字面值 `true` 和 `false`，命令行标记，还有下面表格列出的平台测试函数。你可以指定命令行参数 `-D <#flag#>`。

| 函数 | 有效参数 |
| --- | --- |
| os() | OSX,iOS |
| arch() | x86_64,arm,arm64,i386 |

> 注意 arch(arm) 对于 ARM 64 位的设备不会返回 true。 arch(i386) 当代码作为 32位 >iOS 模拟器编译的时候，会返回 true。

简单的条件编译语句使用如下形式：

```
#if build configuration
    statements
#else
    statements
#endif
```

*statements* 又零个或多个有效的 Swift 语句组成，可以包括表达式，语句，和控制流语句。你可以为条件编译语句添加 `∧∧` 和 `||` 操作符来增加附加的编译选项，使用 `!` 操作符来取反，并且通过 `#elseif` 来添加条件语句块。

```
#if build configuration && !build configuration
    statements
#elseif build configuration
    statements
#else
    statements
#endif
```

和 C 预处理器中得条件编译语句对比，Swift 中的条件编译语句必须是用字包含并且语义正确的代码块进行包围。这是因为所有的 Swift 代码都会被进行语法检查，即使他们是没有被编译的。

更多内容请参看 www.theswiftworld.com 网站

# 用 Objective-C 的行为来写 Swift 类

互操作性让你能用 Objective-C 的行为定义 Swift 类。你可以在写 Swift 类的时候集成 Objective-C 类，采用 Objective-C 协议，和使用 Objective-C 其他功能的特性。这意味着在 Objective-C 熟悉的基础上用 Swift 现代化并且强大的语言特性来加强它们。

## 继承自 Objective-C 类

在 Swift 中，你可以定义 Objective-C 类的子类。要在 Swift 中创建 Objective-C 的子类，在 Swift 类名后面添加一个冒号 (:)，后面紧随着 Objective-C 类的名称。

```
import UIKit

class MySwiftViewController: UIViewController {
    // define the class
}
```

你得到了所有 Objective-C 父类中提供的功能。如果你要提供你对父类方法自己的实现，记得使用 `override` 关键字。

## 采用协议

在 Swift 中，你可以采用定义在 Objective-C 中的协议。就像 Swift 协议，Objective-C 协议也在父类名后面用逗号分隔开。

```
class MySwiftViewController: UIViewController, UITableViewDelegate, UITableViewDataSource {
    // define the class
}
```

如果你要在 Swift 代码中引用 `UITableViewDelegate` 协议，直接用 `UITableViewDelegate` 来引用它(就像在 Objective-C 中用 `id<UITableViewDelegate>` 来引用一样)。

因为 Swift 中没有类的命名空间和协议的定义， Objective-C 中的 `NSObject` 协议在 Swift 中被重新映射为 `NSObjectProtocol` 。

## 创建初始化方法和清理方法

Swift 编译器为了提升你代码的安全性和可预测性，要确保你的初始化方法中对所有的属性都进行了初始化。另外，不像 Objective-C，在 Swift 中没有单独的内存分配供调用方法使用。你要使用原生的 Swift 初始化语法，即便当你使用 Objective-C 类的时候——Swift 将 Objective-C 的初始化方法转化成 Swift 初始化方法。你可以在初始化方法这一章中读到更多实现你自己初始化方法的内容。

当你想在你的类清理的时候执行一些附加的清理工作的时候，你可以实现一个清理方法来代替 `dealloc` 方法。Swift 清理方法会被自动调用，整好在实例销毁之前。 Swift 在调用你子类的清理方法时，也会自动调用父类的清理方法。当你使用 Objective-C 类的时候或者你的 Swift 类继承自 Objective-C 类的时候， Swift 也会为你调用你的父类的 `dealloc` 方法。你可以在清理方法一章中查看更多关于如何自己实现清理方法的内容。

## 与 Interface Builder 集成

Swift 编译器包含了为你的 Swift 类打开 interface builder 特性的属性。就像在 Objective-C 中， 你可以在 Swift 中使用 outlet，action， live rendering。

## 使用 Outlet 和 Action

Outlet 和 action 能让你将你的代码与 Interface Builder 中的用户界面进行关联。要在 Swift 中使用 outlet 和 action，需要在属性和方法定义的前面加上 `IBOutlet` 或 `IBAction` 。同样使用 `IBOutlet` 来定义 outlet 集合——只需要将类型指定为

array。

当你在 Swift 中定义一个 outlet，编译器自动会将这个类型转换成隐式拆包后的 Optional 值，而且是弱引用的，并为它赋初始值为 nil。实际上，编译器将 `IBOutlet var name: Type` 替换为 `IBOutlet weak var name: Type! = nil`。编译器已经将它转换为隐式拆包的 Optional 类型，这样就不必一定要在初始化方法中为它指定初始值。因为在从 storyboard 或 xib 文件初始化后，你可以假设 outlet 已经被连接上。Outlet 默认是 weak 引用，你创建的 outlet 默认都是弱引用关系。

例如，如下的 Swift 代码中的类，定义了一个 outlet，一个 outlet 集合，还有一个 action：

```
class MyViewController: UIViewController {
    @IBOutlet var button: UIButton
    @IBOutlet var textFields: [UITextField]
    @IBAction func buttonTapped(AnyObject) {
        println("button tapped!")
    }
}
```

因为 `buttonTapped:` 方法传入的参数没有被使用，参数名就可以忽略掉。

## Live Rendering

你可以使用两个不同的属性——@IBDesignable 和 @IBInspectable——来打开在 Interface Builder 设计的实时交互性的自定义视图。当你创建一个继承自 `UIView` 和 `NSView` 的自定义视图，你可以在类定义前面加上 `@IBDesignable` 属性。当你将自定义视图添加到 Interface Builder 后(在 inspector 面板中设置视图的自定义类)，Interface Builder 在 canvas 上渲染你的视图。

Live rendering 只能从导入的框架中使用。

你还可以为类的属性添加 `@IBInspectable` 属性。在你将自定义视图添加到 Interface Builder 后，你可以在 inspector 中编辑这些属性。

```
@IBDesignable
class MyCustomView: UIView {
    @IBInspectable var textColor: UIColor
    @IBInspectable var iconHeight: CGFloat
    /* ... */
}
```

## 指定类属性的属性

在 Objective-C 中，类的属性有一系列的特性可以对属性的行为指定一些附加信息。在 Swift 中，你用不同的方式来指定这些特性。

## Strong 和 Weak

Swift 的属性默认情况下是 strong 的。使用 `weak` 关键字来表示属性是一个指向存储为它的值的对象的弱引用。这个关键字只适用于那些 Optional 类型的属性。关于更多信息，可以查看 Attributes 一章。

## 可读可写和只读

在 Swift 中，没有 `readwrite` 和 `readonly` 属性。当定义一个存储的属性时，`let` 表示只读，`var` 表示可读可写。当定义一个计算出来的属性时，只提供 getter 方法，它就是只读的，同时提供 getter 和 setter 方法，它就是可读可写的。更多信息，可以查看 Properties 一章。

## Copy 语法

在 Swift 中， Objective-C 的 `copy` 类型的属性被翻译为 `@NSCopying`。属性的类型必须遵从 `NSCopying` 协议。关于更多的信息，请查看 Attributes 一章。

## 实现 Core Data Managed Object 的子类

Core Data 提供了 `NSManagedObject` 子类属性的底层存储和实现。为你在 Core Data 数据模型中相应的属性或关系的子类的每一个类属性前面加上 `@NSManaged` 标记。就像 Objective-C 中的 `@dynamic` 标记，`@NSManaged` 告诉 Swift 编译器，一个属性的存储和实现将在运行时提供。然而，不像 `@dynamic`，`@NSManaged` 标记，只有在 Core Data 的支持中可用。

Swift 的类是用命名空间管理的——他们用他们编译进的模块来区分(一般情况下是项目)。要使用 Swift 对 `NSManagedObject` 的子类到你自己的 Core Data 模型中，在 Class 文本框中加入你模型的前缀。

更多内容请参看 www.theswiftworld.com 网站

# 将你的 Objective-C 代码迁移到 Swift 中

迁移 提供了通过替换 Swift 代码，修正你的 Objective-C 应用并改进它的架构，逻辑和性能的机会。为了更加直观， 对 app 的增量迁移，你需要使用前面学到的工具 - 混合和匹配加上互操作性。混合和匹配能够更容易的选择在 Swift 中要实现 哪些特性和功能，并且在 Objective-C 中留下哪些功能。互操作性，让这些特性回到 Objective-C 实现更加无障碍。 使用这些工具来探索 Swift 的扩展功能并将他们集成回 Objective-C 应用中，并且不需要对 Swift app 进行任何更改。

## 准备你的 Objective-C 代码用作迁移

在你准备迁移你的代码之前，确保你的 Objective-C 和 Swift 代码拥有最佳的兼容性。这意味着你需要整理并对你现存的 Objective-C 代码进行现代化修改。你现存的代码应该遵从更现代的编码实践，这样能让它更容易的和 Swift 进行无缝交互。关于要采用的实践的简短列表，请查看 "采用现代化的 Objective-C" 一章。

## 迁移过程

将代码迁移到 Swift 中最有效的方法是以逐个文件的方式，就是，每个类使用一个文件。因为你可以将 Swift 的类在 Objective-C 中继承。你可以将这个类的 `.m` 和 `.h` 文件通过一个 `.swift` 文件来替换。你的所有接口和实现都会直接进入这个 Swift 文件中。你不需要创建一个头文件； Xcode 会在你需要的时候自动生成头文件。

## 在你开始之前

- 为你的 Objective-C 的 `.m` 和 `.h` 文件创建 Swift 类，通过选择 File > New > File > (iOS or OS X) > Source > Swift File 来创建。你可以使用和你的 Objective-C 类相同或不同的名称。在 Swift 中，类前缀是可选的。
- 导入相关的系统框架
- 如果你需要在同一个 app target 中你的 Swift 文件中访问 Objective-C 代码的话，还需要填充 Objective-C bridging header 文件。关于如何填充，可以参看 "导入同一个 app target 的代码"。
- 要让你的 Swift 类在 Objective-C 可访问并且可用，就要让他继承 Objective-C 类，或者使用 `@objc` 属性来标记它。如果要对在 Objective-C 使用的类的名称进行特殊指定，需要使用 `@objc<#name#>` 来标记它，<#name#> 代码你 Objective-C 代码中要引用的 Swift 类名。关于 `@objc` 的更多信息，可以查看"Swift 类型兼容性" 一章。

## 你的工作

- 你可以通过继承 Objective-C 类，采用 Objective-C 协议，等等，来让你的 Swift 类集成 Objective-C 的行为。关于更多信息，请查看 "使用 Objective-C 行为来写 Swift 类"。
- 在你操作 Objective-C API 的时候，你需要知道 Swift 如何转义一些 Objective-C 的语言特性。关于更多信息，请查看 "和 Objective-C API 交互这章"。
- 当写与 Cocoa 框架交互的 Swift 代码的时候，记住一些类型是被桥接的，这意味着你可以在使用 Objective-C 类型 的地方，使用 Swift 类型。关于更多信息，请查看 "和 Cocoa 数据类型交互"。
- 如果你要将 Cocoa 的设计模式应用于 Swift 类中，查看 "采用 Cocoa 设计模式" 这章中关于迁移公用设计模式的更多信息。
- 关于将你的 Objective-C 类属性转移到 Swift 属性中的信息，请查看 "Swift 编程语言这章"。
- 在需要的时候对 Objective-C 属性和方法名称使用 `@objc(<#name#>)` 属性。例如， 你可以在 Objective-C 中对一个叫做 `enabeld` 的属性进行标记，来得到叫做 `isEnabled` 的 getter 方法。

```
var enabled: Bool {
@objc(isEnabled) get{
    /* ... */
}
}
```

- 将实例方法(-)和类方法(+)分别用 `func` 和 `class func` 进行标记。
- 将简单宏定义为全局常量，并将复杂的宏定义成函数。

## 在你完成后

- 更新你的 Objective-C 代码的 import 语句(更新为 `#import "ProductModuleName-Swift.h"`),就像 "从同一个 App Target 导入代码" 中描述的。
- 通过反选 target 中的成员复选框，将最初的 Objective-C `.m` 文件从 target 中删除。 不要直接将 `.m` 和 `.h` 文件直接删除；可以用他们来调试问题。
- 如果你对 Swift 类提供了新名字，更新你的代码使用 Swift 类名来代替 Objective-C 类名。

## 调试问题技巧和提示

每个迁移过程体验，根据你的现存代码而不同。然而，还有好多通用的步骤和工具来帮助你来调试你代码迁移中的问题。

- 记住你不能够在 Objective-C 中继承 Swift 中的类。这样，你迁移过来的类不能拥有任何的 Objective-C 子类。
- 当你将类迁移到 Swift 中时，在构建之前你必须将相应的 `.m` 文件删除掉，这样才能 避免重复符号的错误。
- 如果要在 Objective-C 中可访问并可用，Swift 类必须继承自 Objective-C 类，或者用 `@objc` 来标记。
- 当你将 Swift 代码迁移到 Objective-C 中时，记住 Objective-C 不能将 Swift 中一些专有的功能迁移过来。关于 详细信息，可以查看 "在 Objective-C 中使用 Swift" 一章。
- 按住 Command 按键然后点击 Swift 类的类名来查看生成的头文件。
- 按住 Option 键然后点击一个符号可以查看关于它的更多信息，比如它的类型，属性，和文档注释。

更多内容请参看 www.theswiftworld.com 网站